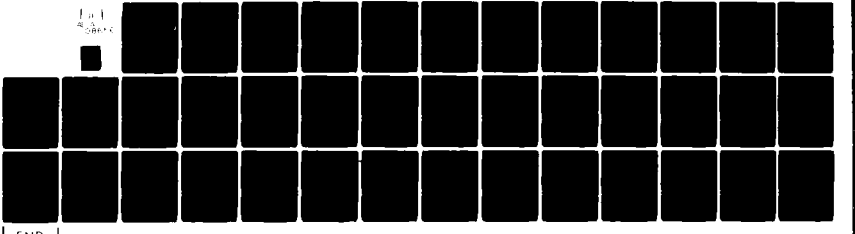
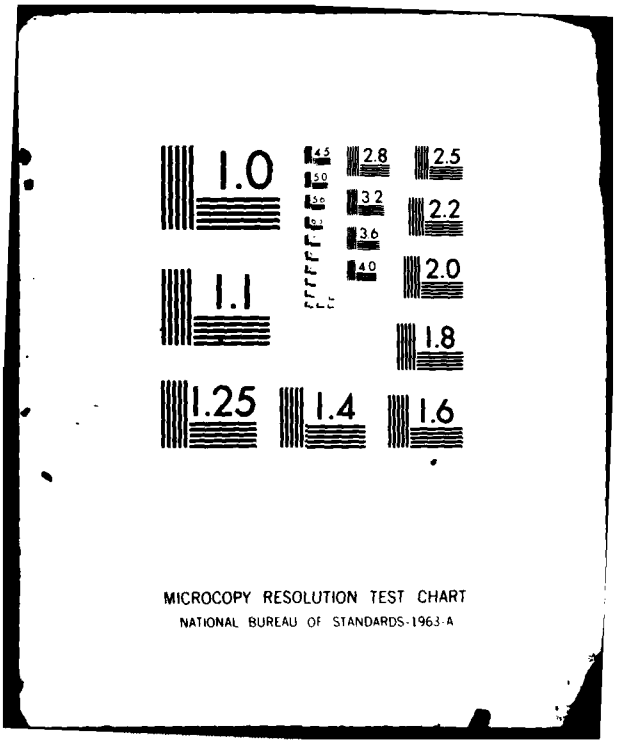
AD-A108 656 DELAWARE UNIV NEWARK DEPT OF COMPUTER AND INFORMATI--ETC F/G 9/2
PRACTICAL ISSUES IN HAVING A USABLE LIBRARY OF SOFTWARE SPECIFI--ETC(U)
MAR 81 R M WEISCHEDEL F49620-79-C-0131
UNCLASSIFIED AFOSR-TR-81-0799 NL



END
DATE FILMED
-82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AFOSR-TR- 81-0799

LEVEL II

AD A108656

Practical Issues in having a usable Library of Software Specifications*

DTIC
DEC 15 1981
H

by

Ralph M. Weischedel
Department of Computer & Information Sciences
University of Delaware
Newark, DE 19711

DTIC FILE COPY

*Research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grants numbered AFOSR-78-3539, AFOSR-80-0190, and contract no. F 49620-79-C-0131. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation herein.

Approved for public release;
distribution unlimited.

81 12 14 060

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 81-0799	2. GOVT ACCESSION NO. AD-A108	3. RECIPIENT'S CATALOG NUMBER 656
4. TITLE (and Subtitle) PRACTICAL ISSUES IN HAVING A USABLE LIBRARY OF SOFTWARE SPECIFICATIONS		5. TYPE OF REPORT & PERIOD COVERED interim technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ralph M. Weischedel		8. CONTRACT OR GRANT NUMBER(s) AFOSR-80-0190
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer & Information Sciences University of Delaware Newark, DE 19711		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB Washington, DC 20332		12. REPORT DATE March, 1981 13. NUMBER OF PAGES 38
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

library of specifications, formal specification methodology, English specifications, documentation, understandability, SPECIAL, abstract data types

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Though formal specifications of software modules offer much toward the design problems of large software systems, creating formal specifications is very difficult, requiring much upfront effort. This paper examines a common idea for dealing with the high cost of software, but in the context of specification. That idea is a pool or "library" of specifications, so that it is easy to build on the work of others. Unlike other efforts that have concentrated on technical problems in having such a library, this paper identifies and studies several common sense requirements on such a library being

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

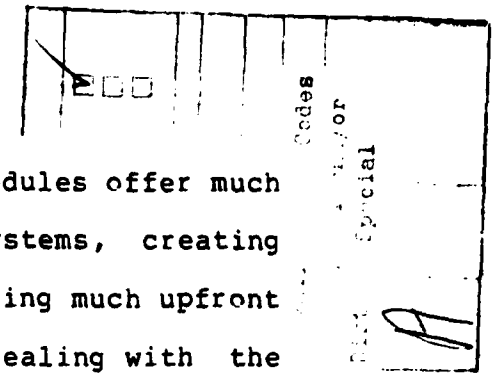
ITEM #20, CONTINUED:

effectively used. Such issues are closer related to human factors than to technical problems; yet these are clearly as critical in use as technical issues.

Our conclusions have arisen from two studies. In one, we wrote several module specifications in the form that might appear in a library. The modules varied in complexity from a stack to the kernel of a text editor; the text editor specifications ranged in length from 9 to 28 pages including copious comments as in-line documentation. The second study compared portions of the English and formal specifications of KSOS (Ford Aerospace, 1978), the Kernelized Secure Operating System.

An example of the kind of library entry we recommend is included in an appendix.

Many of the suggestions are not new ideas, but are merely common sense. In that sense, the paper may be viewed as rather tutorial about writing correct, understandable formal specifications for others to use in their system design.



Abstract

Though formal specifications of software modules offer much toward the design problems of large software systems, creating formal specifications is very difficult, requiring much upfront effort. This paper examines a common idea for dealing with the high cost of software, but in the context of specification. That idea is a pool or "library" of specifications, so that it is easy to build on the work of others. Unlike other efforts that have concentrated on technical problems in having such a library, this paper identifies and studies several common sense requirements on such a library being effectively used. Such issues are closer related to human factors than to technical problems; yet these are clearly as critical in use as technical issues.

Our conclusions have arisen from two studies. In one, we wrote several module specifications in the form that might appear in a library. The modules varied in complexity from a stack to the kernel of a text editor; the text editor specifications ranged in length from 9 to 28 pages including copious comments as in-line documentation. The second study compared portions of the English and formal specifications of KSOS (Ford Aerospace, 1978), the Kernelized Secure Operating System.

An example of the kind of library entry we recommend is included in an appendix.

Many of the suggestions are not new ideas, but are merely common sense. In that sense, the paper may be viewed as rather tutorial about writing correct, understandable formal specifications for others to use in their system design.

1. Introduction

The technique of formal module specifications seems to offer much toward alleviating many problems of large software systems (those systems requiring at least 25 programmers for development and at least 30,000 lines of source code). The high cost of software maintenance, the predominance of design errors, the difficulty in modifying software, and the difficulty and cost of diagnosing and correcting design errors are some of the problems addressed by formal specifications based on the information-hiding principle. Yet, the creating of formal specifications is very difficult, requiring much upfront effort. For instance, Parnas (1976, p. 7) states, "Experience has shown that the effort involved in writing the set of specifications can be greater than the effort it would take to write one complete program."

An oft proposed idea for reducing costs in creating software is to have a library so that specifications, programs, etc. may be reused from previous projects, rather than being created from scratch. Naturally, the mechanism of hierarchically constructing larger and larger module specifications from smaller ones is available in the formal languages (e.g. Guttag, et al., 1978 and Roubine and Robinson, 1976). Furthermore, Cooprider (1979) has investigated what information must be recorded to store a family of versions of a hierarchically defined system.

However, we have found that several practical issues arise if a software designer is to draw on a collection of modules written by others in the course of their designing other

systems. For example, if several general types of text editors exist in the collection, the designer must be able to

a) retrieve the specifications relevant to the need,

b) understand the alternatives quickly to identify which, if any, best suits the current need, and

c) modify the closest alternative, if necessary, to meet the specific needs at hand.

Consequently, not only must each specification in the collection be highly understandable and modifiable, but also the number of logical alternatives for a given task such as text editing must be fairly small. These conditions must hold for effective, timely use of previously written specifications compared to simply creating them from scratch.

We have found in our examples that only a handful of alternatives are necessary to cover a given need, if several principles we found are followed.

1.1 Our Point of View and Use of Terminology

Since the meaning of module, design, specification, and library is often in the eye of the beholder, we outline our use of them here. By module we mean a system or subsystem. This is not the same as an abstract data type (Guttag, 1980), since a compiler is a module but does not define a data type. Hence, all abstract data types are modules, but a module is more general than an abstract data type.

By design we mean the decomposition of a large system into precisely defined modules. In the design methodology advocated (Parnas (1972), Robinson, et.al. (1977) and others), a module

specification defines precisely the interface of a module by detailing exactly what each function at the interface of a module does without committing oneself to any particular implementation.

The kind of library considered here would be a reference collection of module specifications for designers to browse through. For any given need, such as a file management module, the library would contain a number of alternative specifications ranging through the spectrum of functional capabilities a file management module might offer. Ideally, sophisticated tools would be available for a designer to easily knit specifications of various modules together into a complete system specification; however, we have not addressed development of such tools.

Our interest in formal specifications in this paper is in their use as a precise, unambiguous design document detailing the interfaces of modules of large systems. Therefore, our interest here is in their communication aspects among people since designers must select intelligently among alternative specifications in the library.

..2 Basis for Suggestions

Our suggestions arise from two studies. The first involved creating a collection of specifications as might appear in a library. Several data structures including two kinds of stacks, four kinds of queues, and binary trees were specified. In addition, the functional capabilities of three classes of text editors were specified by us. The text editor specifications ranged from 9 to 28 pages including copious comments documenting them. In addition, implementations were written for each of the

modules specified except one of the text editors.

The second study has involved comparing sections of the English specification and the formal specification of KSOS (Ford Aerospace, 1978), the kernel of a secure operating system. One emphasis here has been comparing the means of conveying information, its organization, and the type of information in the formal language and in English. KSOS was chosen, since it is one of the largest, most complex systems that has ever been formally specified and since both types of specifications are available for it.

2. Prospects for a Library of Formal Specifications

There are six issues to consider in the feasibility of such a library. Understandability is so critical that we consider it separately in section 3. The others are discussed in sections 2.1-2.5.

Appendix II presents a definition of a stack module (called stack1) which will serve as an example in our discussion. They are written in SPECIAL (Roubine and Robinson, 1976); a brief description of that language appears in Appendix I.

2.1 Modifiability of a Specification

Given that the most appropriate specification is found, it may not be a perfect match to the designer's needs. In that case, the more easily the specification can be modified to suit those needs exactly, the better. Two commonsense techniques are important.

1) The first is to structure the definitions of each function at the interface of a module so that each particular

detail that might need modification is localized in only one subdefinition. Then only one subdefinition need be changed rather than the changes being spread throughout the specification. (This is just the notion of abstraction.) For instance, suppose one is defining a search operation for a text editor, where the search pattern is a limited form of regular expression. By abstraction, one can give a toplevel definition, localizing the definitions of the syntax and semantics of patterns in a way that makes them very modifiable. The details of this example appear in Weischedel (1979).

2) The author of the specification should list the decisions that are arbitrary and depend on the environment in which the module will be used. As an example in the module stack1, only the top stack element can be removed; the documentation describes the change to make any stack element retrievable. Section E of the documentation in Appendix II lists this information.

These two well-known principles will make the prototypes in the library rather modifiable.

2.2 Number of Prototypes

For any particular type of module needed, the ideal would be that a handful of prototypes would cover the major possibilities for a given need, so that the designer can quickly ascertain which, if any, fits best. If there are many prototypes necessary for each application, then the time spent analyzing each one will make using the library prohibitive.

In our study of data structures and text editors, a handful of prototypes seem adequate. Even if the instances below are

off by a factor of two in the number of variations needed, the fact that only a handful suffice means that the number of prototypes to be examined is not prohibitive.

For a stack, two versions seem necessary: one from which one can read only the last item stored, and one permitting any value designated by a movable pointer to be read, but not modified.

For a queue, we suggest four variations: one where reading occurs at only the front, a priority queue, and two character streams. In a priority queue, the first entered of the largest values (highest priority) is read or removed from the sequence before any others; no other values can be read. Character streams enable the details of synchronizing input/output operations to be hidden in the module rather than forcing all programs to be aware of the means of synchronization. One of the character streams is character oriented; the other is oriented to lines or variable-length blocks of characters.

For trees, we suggest two variations: the binary tree and the general tree with arbitrarily many branches. We do not consider a threaded tree a third logical variation, since it is an implementation of a fast means of performing tree traversal, an operation specified at the interface of the module. Therefore, though implementations using thread links affect performance, they do not change the functional capability.

For text editors, we found three logical alternatives. One has operations oriented to adding, deleting, or moving characters. Another has operations oriented to lines and line numbers. The third example is an editor whose operations are

oriented to moving a cursor on a CRT screen and editing via changing the screen.

Three principles which emerged from specifications we wrote should cut down on the number of alternatives necessary for any given application, if the principles are followed in writing a new entry for the library.

1) Select a consistent set of decisions for those details that depend on the module's use in practice and make those decisions easy to modify using the two suggestions in section 2.1.

2) Avoid issues that are not fundamental to the logical, functional capabilities of the module. Those differences would multiply the number of entries for a given application in the library without adding any new abilities. For instance, in specifying text editors, we did not define a user command language, for there are many legitimate syntactic variations, each of which will be of varying value to different user communities.

3) Specify many fundamental, primitive operations to give each prototype a maximal number of basic features. A designer, after selecting a specification from the library, can delete operations from the interface not needed in his/her environment, assuming that each operation was defined using information-hiding as stated in principle (1). For instance, our specification of a line-oriented editor was patterned after a subset of the functional capabilities of SOS (National Institute of Health, 1977). Operations corresponding to the alter mode, where the user can modify a range of lines using character-

oriented operations relative to a pointer, can easily be removed. A second example is in Appendix II, the stack module with a movable pointer for retrieving values at that position. Six pointer movement operations are included; any may be removed.

In conclusion, we have insufficient evidence to extrapolate beyond the domain of data structures and medium-sized software tools such as text editors. As the size of the module grows, it is not clear that only a handful of prototypes will be sufficient. Nevertheless, the fact that so few sufficed in those two domains is quite encouraging.

2.3 Contents of a Library Entry

First, we consider the type of documentation for any given formal specification in the library; then we consider whether implementations can be stored as well.

The documentation with the entry is critical, not only for understandability, but also for the designer to be able to quickly eliminate entries that are not close to his/her need. Then, the designer can focus attention on two or three that are most promising. Otherwise, the library would bog designers down on issues that should be resolved quickly. Five types of information seem valuable for quickly deciding on the relevance of a module specification. (Appendix II contains this documentation as well as the formal specification.)

1) One is a description of the purpose of the module and the kinds of needs it fills.

2) A second is a summary description of each class of functional capabilities the system has. Notice that this kind

of information is not stated explicitly in formal specification languages, but is implied. For instance, in Appendix II the English summary "Description" states explicitly that the first pointer operation must be either set_pointer_top or set_pointer_bottom. This is implied by the formal specification, though not stated explicitly, since otherwise the pointer is undefined (the "?").

3) Third, the kinds of decisions not made by the module should be described; these are implementation issues which would have to be decided after selecting a module specification, when the coding phase begins. This will reinforce the fact that answers to those issues have not already been given. The section labelled "Hidden Information" in the appendix corresponds to this. For instance, in specifying a text editor, we would write in this section, that a decision to use array storage, linked lists, or other alternatives to store the file being edited, would have to be made when programming of this module begins.

4) Specific references, such as texts and journals, if available, should be given describing various implementations, algorithms, and analyses of them for use when programming begins.

5) As discussed in section 2.1, the author of the specification must include all ways foreseeable that the specification might require modification to tailor it to specific needs. The "Modifications" section in the examples covers this.

In addition to the documentation identified above, storing

implementations would be highly desirable. Even for a module which has a very simple specification, there may be many different implementations. For instance, Horowitz and Sahni (1976) presents a specification of a symbol table; yet, that text spends the majority of two chapters describing and analyzing alternatives for implementation, such as linear search, binary search, fibonacci search, various hashing techniques, trie indexing, etc. Even a single operation can have many competing algorithms; consider sorting with quicksort, heapsort, bubblesort, and insertion sort as alternative implementations.

As one considers larger and larger modules, the number of possibilities could grow dramatically. For the kernel of an operating system, one would have various hierarchical decompositions of the kernel (as a module) into many smaller modules. For each of the smaller modules of each of the decompositions, there would be alternatives in implementation. Some reprogramming may be necessary, since the module specification may oftentimes need slight modification due to varying environments in which the module is to be used. This will require each program stored to be very well structured and very well documented so that it may be easily modified.

It should be technically feasible in the foreseeable future to include with the functional specification of the module's interface various hierarchical decompositions and corresponding programs implementing the module. Cooprider (1979) presents a technique for defining and maintaining a family of software systems. His Software Construction Facility (SCF) provides a module interconnection language which defines module

interconnections of entire systems, the shared aspects (and differences) among versions of those systems, and the sequence of operations for assembling versions of modules into a complete system. Parts of the interpreter for SCF have been implemented. This technique might be adapted to enable a family of implementations to be stored in the library even for large systems. Such tools would be invaluable in a program support environment (Buxton, 1980), since both system specification and system construction would be greatly facilitated.

However, since the number of implementations that may need to be stored for any module could be relatively large, developing a reasonably complete library including implementations as well could take many years. However, the library would be of great value to designers just with the module interface specifications as the alternative programs for each module are added slowly.

2.4 Retrieval

Clearly, facilitating retrieval from the library is a crucial issue for the library to succeed; however we have no conclusions here. It is not clear what assumptions one should make regarding the designer. Should one assume the need will be clearcut, such as needing a specification for a symbol table that is suitable for a block-structured language? Or is it more likely the case that the designer will have only a vague notion of the need and will have to browse among many wide-ranging classes of modules? When it is clear what assumptions can be made about the designer, one can determine whether existing retrieval techniques can be used or modified.

As a practical issue in retrieval, some modules returned for a given request are likely to be rather distant from the true need. The design of the data base of modules and of the retrieval language should minimize this for effective use of the library.

2.5 Correctness

Obviously, the specifications in the library should be correct. A program is correct if it fulfills its specification. One checks program correctness by comparing what the program does against what the specification says it should do. What then does one mean by correctness of a formal specification of a module? By what standard is it deemed correct? Liskov and Zilles (1975) views the specification process as a translation from the concept in someone's mind of what a module should do to a formal specification. *In general*, it must conform to our intent for the module.

One could define correctness of specifications in terms of writing two different specifications and proving them equivalent, but this misses the point of the difficulty of writing specifications. Rather, we prefer an informal notion of correctness of a specification. To be correct, it must conform to the intent for the module, it must be internally consistent, it must leave no cases unspecified, and it should rule out implementations that do not conform to the intent for the module. Guttag and Horning (1978) presents formal definitions of consistency and completeness. Though their definitions are given for languages based on algebraic axioms, it is easy to define similar notions for other types of specification languages.

Gerhart and Yelowitz (1976) presents two examples of formal specifications which were incorrect in that they left out a crucial feature; therefore, programs could be written that fulfilled those specifications but which did not satisfy the person's intent.

Three obvious techniques proved a significant aid to us in checking the correctness of specifications.

1) Software tools should perform as many checks as possible. Development of such tools is an active area of research, and several examples exist. (e.g. Roubine and Robinson, 1976; and Musser, 1980). We used the SPECIAL specification handler (Roubine and Robinson, 1976) which verifies syntactic correctness and performs type checking on all expressions in SPECIAL, a strongly typed language.

2) Peer review uncovers many errors, as well as offering valuable comments for improving understandability. If one creates documentation according to our guidelines in section 3, the documentation will provide much detailed insight regarding the intent of the specification, and therefore will provide a basis for judging correctness.

3) If one is not using an operational specification language, and if the software tools available cannot simulate the module specified, then a quick implementation in a very high level language will provide a concrete system for testing the functional capabilities of the module in question. In general, we wrote implementations in INTERLISP (Teitelman, 1975) for our SPECIAL specifications. This often uncovered specification errors not detected by peer review or the specification handler.

The implementations, since they were in such a very high level language, took remarkably little time, for using the very high level language enabled us to trade performance characteristics of the test module for programmer time. For instance, the final module checked in this way took only two to three days for one programmer to implement, even though it was a kernel providing the functional capabilities of a character-oriented text editor. Furthermore, we found significant regularity in the implementation of most types of expressions in SPECIAL, suggesting that much of each implementation could be done automatically by a software tool.

3. Understandability

Researchers in the area of formal module specifications and abstract data types generally agree that they are difficult to understand, though the degree of difficulty is argued.

The formal specification of a module must be understandable if it is to achieve its purpose, for it acts as a contract between designers and programming team, stating exactly what the programming team's product must do (Parnas, 1977). Unless they are understandable, 1) programmers will not know what the module they are to implement is to do nor how to use other modules, and 2) designers will not be able to detect design errors nor easily confirm that their design satisfies user requirements. Also, if one is to use a reference library of formal specifications, they must be understandable, for if the designer cannot understand the alternative specifications, how can an intelligent choice be made among the alternatives? If formal specifications of module interfaces are to become widely

used, they must be understandable.

In 3.1 we list several reasons why formal specifications seem difficult to understand. In 3.2 we make concrete suggestions, particularly regarding documentation of the library entries, which will aid understandability.

3.1 Causes of lack of understandability

We have found several reasons for the difficulty of understanding formal specifications, particularly as compared to natural language specifications. These observations arise primarily from our study (described in section 1) of the English and formal specifications of KSOS.

Two of the observations are well-known, dealing with the nature of formal specification languages.

1) Formal specifications usually contain far more detail than natural language ones do. Attention to detail, of course, is requisite in specification. Liskov and Berzins (1977) agree, stating on p. 13-5, "Rigorous informal specifications are probably just as difficult to construct as formal ones; informal specifications appear easier to construct because they are usually incomplete."

2) The semantics of specification languages is often quite different than programmers are used to. Formal specifications of modules are to be implementation independent. Programming languages are designed to define implementation detail. Thus, the purpose and focus of attention of specification languages are often quite different from programming languages. (The operational specification languages are an exception, since they have semantics similar to programming languages. However,

Guttag (1980) argues that they have additional handicaps for understandability: the irrelevant implementation detail and the need to infer the relations between functions at the interface.)

Observations 3 through 5 concern the nature of natural language.

3) Natural language has myriads of concepts already defined and familiar to us for succinctly stating what a module does, but formal specifications do not as yet. For example, the concepts of a sorted sequence, a pointer, a line of text, and a shared segment of memory are all well-known and are referred to without further explanation. (Yet, this is simultaneously a serious drawback to natural language specifications, since the notion raised in each person's mind may not be standard.) There is no corresponding body of defined concepts which have been taught us and which we have frequently used. Therefore, concepts such as sorted order must be defined in the specification, thus adding to what must be understood. To the reader of a specification, English may appear like a very high level language, whereas the formal language appears like an assembly language without any significant collection of macros or subroutines to draw on. Of course, a library of formal specifications would provide a body of past experience to study and use as in natural language.

4) Natural language specifications can draw on known concepts through analogy; no formal specifications allow this. In the English description of KSOS, frequent analogies are made to UNIX, both to explain features that are similar and to draw specific contrasts; it seems to be a very effective tool there.

Another example is in Horowitz and Sahni (1976). After a lengthy definition of an "ordered list", which is their term for a finite sequence, they introduce stacks by stating (p.77), "A stack is an ordered list in which all insertions and deletions are made at one end, called the top." Some attempts have been made to include analogy in artificial intelligence languages (Bobrow and Winograd, 197?), but no attempt has been made in specification languages.

5) A person's conceptual view of a module is often stated in much different terms than that of present specification languages. An example of this is a spatial view of modules. The statement introducing a stack in the previous paragraph uses the terms 'end' and 'top', clearly indicating a spatial view of a stack, rather than a purely mathematical view in terms of sequences. Hobbs (1977) states that many descriptions of algorithms use a spatial view.

Observations 6 and 7 deal with the different organization of information in formal specifications compared to natural language.

6) English specifications often provide summaries of detail, whereas formal ones have not, as yet, tending to provide detail in an isolated way. For instance, in the KSOS specification (Ford Aerospace, 1978), the following summarizing statement appears on page 9, "A SEID shall be returned as the result of new object creations (i.e. K_create, K_build_segment, K_create_device, K_fork, and K_spawn)." This summarizes several things: 1) that the five functions listed are the ones that create new objects and 2) that all of them return a SEID as a

value. In the formal specification of KSOS, for each function, the type of the value returned, the inputs, input assumptions, and side-effects are given with each of the functions individually. However, there is no summarizing of either of the two facts as in the English; one must infer this information from the particulars. Understanding seems to require a "cognitive framework" around which to organize particulars; this can be stated in English, but formal languages provide no such organizing conceptual view.

7) English specifications often explicitly state facts which are only mathematically implied in formal specifications. For instance, the English specification of the operation K_build_segment in KSOS goes beyond the description of only that function by spelling out the sequence of other KSOS functions to call to set up shared segments of execute-only code. While one may be able to infer all of the steps from the formal specification, the need to determine so much that is implicit decreases the understandability of the formal specification. Similarly, in the documentation of stack1 (Appendix II), we state that the pointer into a stack for arbitrary retrieval of elements must be set initially by one of two operations. This is implied by the formal specification but not stated explicitly. Algebraic axioms might appear not to have this problem, since they focus attention on the relations between operations at the interface of a module. Yet, examples specified using algebraic axioms can lack understandability also by leaving crucial facts implicit. For instance, a key to understanding the specification of a stack in Guttag (1980) is recognizing that any time that the

stack is empty is equivalent to the time when it was first created. The axioms do not state this explicitly, though one could presumably add one or more axioms to do so. Rather, one must first hypothesize the fact and then prove it true from the axioms.

These observations lead us to our practical suggestions for writing more understandable formal specifications.

3.2 Practical Suggestions for Understandability

The first two suggestions come directly from programming methodology. It is interesting that principles developed for structured programming apply equally well to nonprocedural specification languages of the axiomatic type. This means that the principles deal not so much with managing control flow as with managing detail.

1) Complex definitions should be broken up into short definitions which can be analyzed and reviewed in a top-down way.

2) Long, descriptive mnemonics, such as "max_number_of_stacks," are critical to understandability.

Regarding specification languages, we make the following suggestion.

3) The specification language should provide a rich set of primitive objects and operations. This will alleviate the lack of previously defined concepts. For instance, suppose we are specifying a module which among other things, sorts a sequence. The definition would be much shorter and clearer if a concept permutation(a,b) were already defined in the language or in a library. For that matter, the concept of a sorted sequence is

so common, that its precise definition should be primitive to the specification language. Unless the languages have a rich set of structures for designers to draw on, the amount of detail could be overwhelming.

Suggestions 4 through 8 deal with documentation. Though English will be vague, incomplete, and ambiguous, it conveys a toplevel view around which the complete, unambiguous, precise, formal description can crystallize into understanding in the reader's mind. Of course, the formal specification alone is the arbiter of all questions about the module.

4) An English description of the purpose of a module and of each function available at the interface of the module provides a general notion or conceptualization for understanding the formal specification. Such a high level description is essential documentation for management personnel.

5) The principle for deciding whether to include a comment for a line of the formal specification is whether its purpose and implications would be obvious to the average reader without a comment. (We are indebted to David Crocker for stating this criterion regarding our use of comments in the specifications we had written.) Not only will following (4) and (5) make formal specifications more understandable, but also following them gives each reader of the specification the ability to verify that every aspect and subformula of it corresponds to the author's intent. This is an informal means of design validation; for a library of such specifications, the means is very powerful since more and more designers will be reading and verifying a specification as time goes by.

6) Certain English constructions can be very difficult to understand and should be avoided. The English specification of KSOS is generally well-written and well-organized. However a cryptic style of omitting words in defining variables or of overusing parenthesized descriptions makes understanding difficult on certain points. An example exhibiting both features is the following definition (p. 27 of Ford Aerospace, 1978) of an exception condition of one of the functions: "unable to create new process (possible information channel)". It is unclear to us whether a new process cannot be created because of a possible information channel or a new process which is a possible information channel cannot be created or whether something altogether different is intended. One can avoid these two features simply by stating everything as complete thoughts in complete sentences. For the example, this could be "A new process cannot be created because of a possible information channel".

7) All but very small specifications need an index. To pick an arbitrary figure, formal specifications whose complete hierarchical definition is at least six pages need an index. The parser for the formal specification can easily create this. English ones at least twelve pages in length also need an index.

8) If an English description accompanies the formal specification, as opposed to being embedded within the body of the formal one as comments, then a cross-reference between the two is needed. The cross-reference will not only aid understandability by relating the two but will also provide an informal means for people to check that the formal specification does what its authors claim.

These suggestions are not new ideas, but rather are merely common sense. We have concluded that entries for a library of formal module specifications should be prepared with so much care for understandability that casting away the first attempt at a specification to create a more understandable one is not frowned upon. Peer review can be very valuable to check not only the correctness of a specification but also its understandability.

4. Conclusions

Our preliminary analysis shows that a library of formal specifications is quite promising. It is particularly encouraging that for elementary data structures and for text editors, a handful of alternatives covered the major variations.

Several research topics are called for. One is a characterization of the kinds of requests a designer might have, so that appropriate retrieval techniques could be developed. Second is the implementation of software tools that would make the library part of a comprehensive design environment. A third is development of software tools for integrating stored alternative implementations for a specification into a program support environment (Buxton, 1980). A fourth is in the area of formal specification languages. Since analogy plays such a useful role in natural language specifications, a form of analogy in formal specification languages would be a powerful aid in shortening specifications without sacrificing detail.

Appendix I

Brief Description of SPECIAL

This brief introduction is provided because of the example in Appendix II; it is not intended as a comprehensive description. We will use all capitals when referring to reserved words.

SPECIAL uses preconditions and postconditions to specify each operation which is available at the interface of a module. Operations are called FUNCTIONS. There are three types of functions: OVFUN functions return a value and have side-effects. OFUN functions have side-effects but return no value. VFUN functions have no side-effects, but do return values.

For each function, preconditions are implied by the sequence of EXCEPTIONS. The exceptions are to be checked in the order given; if any are true, the function exits without execution, but with an indication of which exception is true. Therefore, the precondition is that none of the exceptions are true.

For each function, postconditions are listed as EFFECTS. There is no order to the effects; they are simply true statements. For a function that returns a value, one of the effects is, of course, the statement of the value returned. The "=" symbol means equality, not assignment. Oftentimes one must distinguish between the old value that a second function returns and the new value that it returns as a result of the side-effects of the first function. One can refer to the new value by placing an apostrophe before the second function's name. That is, the expression 'f(a)=f(a)+1 in the effects of a

function g means that executing g changes f on the particular value a; f will return a value one greater than what it used to when called with a as its argument.

The arguments of a value-returning function are given by a form such as

function_name (type arg,arg ; type arg ;...) -> type value.

The types of course declare the argument types and the type of the value.

Sometimes it is convenient to define auxiliary functions which are used in the specification but are not part of the module's interface (and possibly will never appear in the implementation). These are defined as HIDDEN. Some VFUNS return a value which is totally defined as an expression of other functions; these may have a DERIVATION stating the expression rather than an EFFECT defining the value returned. All other VFUNS must have an initial value in their definition; this follows the word INITIALLY.

Though the section FUNCTIONS is central to the specification, there are several additional sections. If an expression (or several closely related expressions) appear frequently in the specification, one may state the expression in one place as a DEFINITION as a means of abstraction. The form is

name (typed_arguments) IS expression.

The name and argument list may be used anywhere that the expression should appear.

Though one can give type declarations with the formal arguments themselves, one may declare a type associated with an

identifier throughout the specification. This is accomplished using the DECLARATIONS section, which contains a line of the form:

type identifier1,identifier2,...,identifiern.

One may define new types in terms of others using the TYPES section. This has the form

user_type: definition.

Defining a new user type as a DESIGNATOR is a means of getting a set of names to uniquely identify objects. In our example the type stack is defined as a designator, since we are specifying a module to dynamically manage a set of stacks, and since a name for each stack is necessary.

Oftentimes in defining a module, one needs a constant; yet, that system constant may differ from one system (when actually assembled) to another. One can declare such system constants in the PARAMETERS section. In a specification, only its type need be given.

Expressions include the normal arithmetic and boolean types. A question mark represents the special value UNDEFINED. Expressions may be quantified. For instance,

(EXISTS INTEGER j | c(j) : p(j))

means that there exists an integer j with c(j) being true such that p(j) is true. The contents of a vector may be defined by an expression such as

VECTOR(FOR m FROM initial TO stop : f(m)).

That expression defines a vector containing (stop-initial+1)

elements. The first element of the vector is given by f(initial), and the last by f(stop). The empty vector is written as VECTOR(). The ith element of a vector v is referred to by v[i]. Only two operations of the expression language can be performed on DESIGNATORS. One is the equality check. The second is the operation NEW(d), which will give as a value of a new designator from the class d of designators.

Comments are enclosed by parentheses and preceded by a dollar sign.

Appendix II

The Library Entry for One Stack Module

Stackl

A. References

Chapters 3 and 4 of Fundamentals of Data Structures, by Ellis Horowitz and Sartaj Sahni, Computer Science Press, Inc., 1976 give three implementations of the basic features of a stack. These would have to be extended to allow the pointer operations of module Stackl.

B. Hidden Information

Users of the module cannot answer the following questions: Are the stacks implemented using an array or linked list? Do all stacks share one array or does each have its own array? How many words, bytes, or bits are used per data element in the stack?

C. Description (an aid to understanding the definition, though the definition is the arbitrator of all issues or questions raised)

The following two paragraphs mention features that this module has in common with the module "stacks". The last paragraph deals with features not found in the simpler alternative, "stacks".

This module manages any number of stacks up to the implementation constant "maxstacks". The data structure represented by a stack maintains a sequence of items. "Push" adds an item to the sequence at one fixed end of the sequence. An item may be removed from the sequence at that same fixed end using "pop,"

which additionally gives its value as the value of the procedure call. One may obtain that value without removing the data item from the sequence via "top". One can ask whether there are any elements in the sequence or not, via "empty". The maximum length of any sequence is "maxsize". The data items are integers whose absolute value is bounded by "maxelement". New stacks may be created and old stacks released via "create_stack" and "delete_stack".

The function "stack(s)" is HIDDEN. Therefore, it can never be called, nor does it imply an implementation using arrays or sequential memory. It merely indicates the effects of "push" or "pop" on the sequence of items. The specification implies that only the element most recently entered may be removed; this property has led to the phrase "last-in-first-out" or LIFO.

One can retrieve values from any position in the sequence, using "pointer", a HIDDEN function indicating which is the current position. "Value_pointer" retrieves the value at that location. One can move the current position via any of the operations "find_element_up", "find_element_down", "move_pointer_up", "move_pointer_down", "set_pointer_top", and "set_pointer_bottom". Initially, for any stack, there is no current position; the only way to initialize the current position is with "set_pointer_top" or "set_pointer_bottom". One may delete all elements that were added after a certain item by first positioning the pointer to the first element after the item and by calling "upper_delete".

D. Modifications

Stacks of course do not have to be sequences of integers.

The declarations which must be changed for REALs, CHARs, etc. are marked by comments in the specification.

One may wish to retrieve any element in the sequence, regardless of the pointer's current position. For this, one may add another function:

```
VFUN value(s;INTEGER j)->i;
  $(random access read of data
    items in stack)
  EXCEPTIONS
    stack(s)=?;
    j<1 OR j>size(s);
  DERIVATION
    stack(s)[j];
  $(Note that value(s;1) is the
    first element added to the
    sequence, not the last one.)
```

E. Alternatives

In some applications one may not need the notion of a "current position" and retrieving the value there. Refer to the specification "Stacks" for a module not having this feature.

```
MODULE stack1

    TYPES

stack_name:  DESIGNATOR;
stack_content:  INTEGER;  $( This is for a stack of integers. The
                           type specification must be changed for a
                           different type of data element)

    DECLARATIONS

stack_content i;
stack_name s;
VECTOR_OF stack_content q;
INTEGER p, k;
BOOLEAN b;

    PARAMETERS

INTEGER maxsize,  $( This is the maximum size of any stack)
        maxstacks,  $( This is the maximum number of stacks
                       permitted)
        maxelement;  $( This gives the maximum absolute value
                        storable in any stack. For a stack of data
                        type other than INTEGER, this must be
                        changed)

    DEFINITIONS

INTEGER nstacks
   IS CARDINALITY({ stack_name s | stack(s) ~= ? });
INTEGER size(s) IS LENGTH(stack(s));

    FUNCTIONS

VFUN stack(s) -> q;  $( This represents stack s)
    HIDDEN;
    INITIALLY
       q = ?;

VFUN pointer(s) -> p;  $( Pointer(s) defines a current element in
                          stack s)
    HIDDEN;
    INITIALLY
       p = ?;

VFUN empty(s) -> b;  $( The function returns true, if stack s
                       contains no elements, otherwise, false)
    EXCEPTIONS
       stack(s) = ?;
    DERIVATION
       size(s) = 0;
```

```
VFUN top(s) -> i;
          $( This returns the value most recently pushed onto
           stack s.  Another derived VFUN, which is just the
           macro size(s), might be added)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
    DERIVATION
       stack(s)[size(s)];

OFUN find_element_up(s; i);
          $( The pointer for s is moved to the element, whose
           value is i and which was the first such pushed onto s
           after the current element)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
       pointer(s) = ?;
       NOT(EXISTS INTEGER j | j > pointer(s) AND j <= size(s):
              stack(s)[j] = i);
    EFFECTS
        'pointer(s)
       = MIN({ INTEGER j | j > pointer(s) AND j <= size(s)
                           AND stack(s)[j] = i });

OFUN find_element_down(s; i);
          $( The pointer for stack s is set to the element, whose
           value is i and which was the last such pushed onto s
           before the current element)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
       pointer(s) = ?;
       NOT(EXISTS INTEGER j | j >= 1 AND j <= pointer(s):
              stacks(s)[j] = i);
    EFFECTS
        'pointer(s)
       = MAX({ INTEGER j | j >= 1 AND j <= pointer(s)
                           AND stack(s)[j] = i });

OFUN move_pointer_up(s; k);
          $( The pointer for stack s is set to the element, which
           was pushed onto the stack k elements after the current
           element)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
       pointer(s) = ?;
       k < 0;
       size(s) < pointer(s) + k;
    EFFECTS
       'pointer(s) = pointer(s) + k;
```

```
OFUN move_pointer_down(s; k);
          $( The pointer for s is set to the element, which was
           pushed onto stack s, k elements before the current
           element)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
       pointer(s) = ?;
       k < 0;
       k >= pointer(s);
    EFFECTS
       'pointer(s) = pointer(s) - k;

OFUN upper_delete(s);
          $( All elements which were pushed onto the stack s,
           on or after the current element, get deleted.  The
           pointer for s will still point to the top of the stack,
           unless upper_delete empties the stack (in which case
           the pointer becomes ?.))
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
       pointer(s) = ?;
    EFFECTS
       IF pointer(s) = 1
         THEN 'stack(s) = VECTOR() AND 'pointer(s) = ?
         ELSE 'stack(s)
             = VECTOR(FOR j FROM 1 TO pointer(s) - 1
                      : stack(s)[j])
          AND 'pointer(s) = pointer(s) - 1;

OFUN set_pointer_top(s);  $( The pointer for s is set to the ele-
                           ment most recently pushed onto stack s)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
    EFFECTS
       'pointer(s) = size(s);

OFUN set_pointer_bottom(s);  $( The pointer is set to the element
                              least recently pushed onto stack s)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
    EFFECTS
       'pointer(s) = 1;

VFUN value_pointer(s) -> i;  $( The value returned is the one the
                              pointer currently references)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
       pointer(s) = ?;
    DERIVATION
       stack(s)[pointer(s)];
```

```
OVFUN create_stack() -> s;  $( This initializes a new stack
                               which will be named s.)
    EXCEPTIONS
       nstacks >= maxstacks;
    EFFECTS
       s = NEW(stack_name);
       'stack(s) = VECTOR();

OFUN delete_stack(s);  $( This removes everything from stac.  .
                          Afterwards s cannot be referred to.)
    EXCEPTIONS
       stack(s) = ?;
    EFFECTS
       'stack(s) = ?;

OFUN push(s; i);
         $( Push adds i to stack s, making i the element returned
          by top(s))
    EXCEPTIONS
       stack(s) = ?;
       size(s) = maxsize;
       i < (- maxelement OR i > maxelement;
           $( This must be changed for a data type other than
            INTEGER)
    EFFECTS
        'stack(s)
       = VECTOR(FOR j FROM 1 TO size(s) + 1
                : IF j <= size(s) THEN stack(s)[j] ELSE i);

OVFUN pop(s) -> i;
          $( This removes the item most recently pushed onto s and
           updates the pointer for s so that if it did point to the
           element being popped, it will then point to the new top)
    EXCEPTIONS
       stack(s) = ?;
       empty(s);
    EFFECTS
       i = top(s);
        'stack(s)
       = VECTOR(FOR j FROM 1 TO size(s) - 1:  stack(s)[j]);
       IF pointer(s) = size(s)
         THEN 'pointer(s) = pointer(s) - 1
         ELSE 'pointer(s) = pointer(s);

END_MODULE
```

Acknowledgements

Daniel Chester, David Crocker, Peter Freeman, Leon S. Levy, Jim Neighbors, Linda Salsburg, and Robert Vollum made many valuable suggestions in wading through early drafts of this. Linda Salsburg wrote many of the specifications and quick implementations that influenced the conclusions here.

References

Bobrow, Daniel G. and Terry Winograd, "An Overview of KRL, a Knowledge Representation Language," Cognitive Science, Vol. 1, #1, 1977, 3-46.

Buxton, J. N., "An Informal Bibliography on Programming Support Environments," SIGPLAN Notices, Vol 15, No. 12, 1980, 17-30.

Cooprider, Lee W., "The Representation of Families of Software Systems", Ph.D. Dissertation, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1979.

Ford Aerospace, "Secure Minicomputer Operating System (KSOS): Computer Program Development Specifications (Type B-5)," Technical Report No. WDL-TR7932, Ford Aerospace & Communications Corporation, Palo Alto, CA, 1978.

Gerhart, S. L. and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, 1976, 195-207.

Guttag, J., "Notes on Type Abstraction (Version 2)", IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, 1980, 13-23.

Guttag, John V., Ellis Horowitz, and David R. Musser, "Abstract Data Types and Software Validation," CACM, 21, No. 12, 1048-1063, 1978.

Guttag, J. V. and J. J. Horning, "The Algebraic Specification of Abstract Data Types", Acta Informatica, Vol. 10, 1978, 27-52.

Hobbs, Jerry R., "What the Nature of Natural Language Tells Us About How to Make Natural-Language-Like Programming More Natural", SIGPLAN Notices, Vol. 12, 8, 1977, 85-93.

Horowitz, Ellis and Sartaj Sahni, Fundamentals of Data Structures, Woodland Hills, CA: Computer Science Press, Inc., 1976.

Liskov, Barbara H. and Valdis Berzins, "An Appraisal of Program Specifications," Proceedings of the Conference on Research Directions in Software Technology, Peter Wegner, Jack Dennis, Michael Hammer, and Daniel Teichroew (eds.), 1977.

Liskov, B. H. and S. N. Zilles, "Specification Techniques for Data Abstractions", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, 1975, 7-18.

Musser, David R., "Abstract Data Type Specification in the Affirm System," IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, 1980, 24-31.

National Institutes of Health, "SOS (An Advanced Line-Oriented

Text Editor) User's Guide," Comp. Center Branch, Div. of Computer Research & Technology, National Institutes of Health, Bethesda, MD, 1977.

Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM, Vol. 15, No. 12, December, 1972, 1053-1058.

Parnas, D. L., "On the Design and Development of Program Families," IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March, 1976, 1-8.

Parnas, David L., "The Use of Precise Specifications in the Development of Software," Information Processing 77, B. Gilchrist, (ed.), North-Holland Publishing Company, New York, 1977.

Robinson, Lawrence, Karl N. Levitt, Peter G. Neumann, and Ashok R. Saxena, "A Formal Methodology for the Design of Operating System Software," Current Trends in Programming Methodology, Volume 1, Software Specification and Design, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

Roubine, Olivier and Lawrence Robinson, SPECIAL Reference Manual, Technical Report CSG-45, Stanford Research Institute, Menlo Park, CA, August, 1976.

Teitelman, Warren, "Interlisp Reference Manual," Xerox Palo Alto Research Center, Palo Alto, CA, 1975.

Weischedel, Ralph M., "A Tutorial Example on Writing Understandable Formal Specifications of Software Modules: An Extended Abstract", Proceedings of Micro-Delcon '79, 1979, 104-112.


FILMED